Methods and Models for Combinatorial Optimization, Lab Exercise Report

Alberto Nicoletti

A.Y. 2023/2024, University of Padova

To compile and run part I:
make
./main data/tsp_instance.dat
To compile and run part II:
make
./main.exe data/tsp_instance.dat parameters.dat

1 Part I

1.1 The problem

The exercise requires to implement a given model for TSP (travelling salesperson problem), considering its practical application: calculate the sequence of holes on a board to be done from a drilling machine such that the total amount of time is minimized.

1.2 The implementation

The file *main.cpp* implements the linear programming model provided using Cplex. The structure of the code has been provided by prof. De Giovanni, as well as the *cpxmacro.h* and the *makefile* files.

This is the (simplified) outline of the program:

- Read from file the number of nodes and the weighted adjacency matrix.
- Create the variables (columns in Cplex API) x's and y's. The variable names are stored as *x_i_j* and *y_i_j* and are created one by one using a map structure to keep track of their position.
- Add the constraints (rows in Cplex API).
- Compute the optimal solution and get the time to find it.
- Print the names and values of the variables for the optimal solution. For readability reasons only the variables corresponding to the edges that are part of the solution are printed (i.e. variables with values different from zero).
- Print the value of the objective function.
- Print the time needed to find the optimal solution. Note that this time considers only the optimization part, not the set up of the problem or the printing phase.

1.3 The results

The program correctness has been tested manually for graphs of size 4, 5 and 6.
All the following tests have been done in the computers of LabTA in Torre Archimede. The instances of the problem have been created using the program *cdata.cpp*. For each instance, five different version of it have been generated and tested. Each of the shown time results is the average of the different tests on the five instances.
Two variables have been considered to make the tests: number of nodes and the shape of the instances. Three shapes for the instances have been used, from the most general to the most similar to our real world scenario.

Random graph

Given the number of nodes, the program assigns randomly an integer weight to each edge. The tests showed that the range of values have a big impact on the performances, on the same level of the number of nodes. Finding a solution for a graph with random weights between 1 and 100 can take even 10 times more time than finding a solution for a problem where the range is 1 to 10.

It is important to note that these instances are very general and random and don't reflect very well the real problem. Test results are shown in Table 1.

Results Maximum number of nodes to find a solution with a weight range [1,100] in less than:

0,1s : Max $\sim$ 10 nodes.

1s : Max $\sim$ 50 nodes.

10s : Max $\sim$ 80 nodes.

	Time		
Number of nodes	Range [1,10]	Range [1,100]	Range [1,1000]
10	0.052s	0.067s	0.110s
20	0.242s	0.253s	0.33s
50	1.6s	2.89s	2.6s
80	1.98s	15s	22s
100	2.215s	64s	63s
120	5.62s	121s	179s
150	8.36s	341s	448s

Table 1: Results for random graphs.

Random grid graph

The program creates (implicitly) a grid and randomly places the nodes on points of that grid. The weights are calculated as the euclidean distance between two nodes as if they were two points in the Cartesian plane. For example, for the graph in Figure 1, the edge (5,4) has weight 3 and the edge (0,3) has weight $\sqrt{5}$.

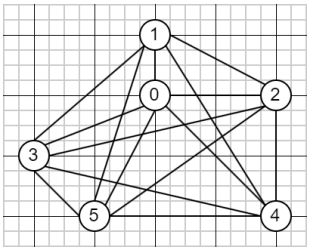

Figure 1: Random grid graph of 6 nodes.

Results Using the same number of nodes but changing the dimensions of the grid, can have a big impact in performances. This brings an high level of uncertainty, so is not possible to provide relevant results about up to which size of the graph the solution is found efficiently. See the different grids for a 50 nodes graph in Table 2 for an example.

Number of nodes	Time
10 (grid 8x10)	0.0711s
20 (grid 8x10)	0.382s
50 (grid 12x10)	4.86s
50 (grid 6x14)	6.47s
50 (grid 100x5)	104s
80 (grid 12x10)	24s
100 (grid 14x18)	131s

Table 2: Results for random grid graphs.

Realistic graph

This is the case where the shape of the instances is the **closest to the real case** of the boards to be drilled, because of the (assumed) regularity of the position of the holes to be made on the boards.

This instances are generated by creating (implicitly) a grid and then by placing, randomly in the grid, groups of four nodes in a rectangular shape of small but random dimensions. Figure 2 shows a graph of 20 nodes on a grid of 22x28.

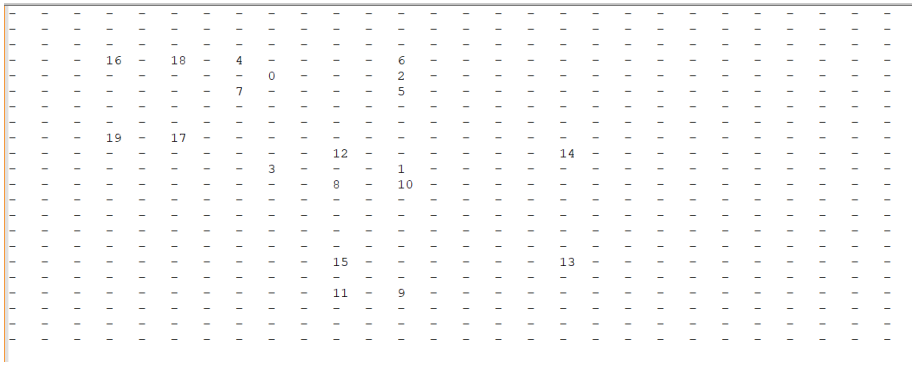

Figure 2: Realistic graph of 20 nodes in groups of rectangles.

Results The tests are made on two different instances of the problem with five different versions each (as for all the other tests). In Table 3 shows the average times along with the single results because of the interesting high time difference between each other. It's important to emphasize how one (relatively small) instance of the problem took more than one hour.

Number of nodes	average time	tested times
50 (grid 300x180)	857s	3691s, 35s, 14s, 240s, 304s
80 (grid 400x3000)	149s	391s, 80s, 165s, 57s, 53s

Table 3: Results for random grid graphs.

1.4 Conclusion

The tests show poor performances, even on graph of relatively small size. This is unsurprisingly because the tsp is an NP-Hard problem.

The tests made on the realistic instances of the problem show bad performances but also high unpredictability, even on different random versions of the same instance of the problem. The number of tests made is small, but good enough to say that this program is not suitable for a practical use in the given real world case, even if we assume the considered instances of the tests as unlucky worst-case examples.

2 Part II

2.1

The problem The exercise requires to design and implement an ad-hoc optimization algorithm for the TSP applied in the real world scenario of Part I, as an alternative to solving the implemented model with Cplex.

2.2 The implementation

For this second part, I have implemented three different heuristic methods, all of them are local searches: Simple Search, Simulated Annealing and Tabu Search. There are some common components along these methods:

- They all use a multistart technique in which the first initial solution is the solution provided by the Christofides algorithm and the other initial solutions are randomly generated.
- The solution representation is the list of ordered visited nodes as a vector of integers.
- The neighbourhoods are 2opt or 3opt.
- The evaluation of the neighbours is the objective function of the tsp (the sum of the weights of the edges in the path).

Moreover, each one of the local searches uses two methods to provided a lower and an upper bound for this solution: 1-tree lower bound and the Christofides algorithm. These bounds could be useful in the real world scenario because they can provide, in (very fast) polynomial time, a range (upper and lower bound) of the time needed to drill each type of board.

The following paragraphs describe the algorithms used for the bounds and the heuristic methods implemented.

1-tree lower bound Given a complete graph *G(V,E)* with V set of verticies and E set of edges, a 1-tree is a lower bound to its optimal TSP solution, that can be found as follows:

- remove any vertex v from V: you get *G'(V\{v},E')*;
- compute the Minumum Spanning Tree T (for example, with Prim's algorithm) on the remaining graph *G'*;
- add back the removed edge v to T connecting it using the two minumum edges in E;

The lower bound used by the programs is the maximum value between all the possible 1-tree in the provided graph.

Christofides The Christophides algorithm is a $\frac{3}{2}$-approximation algorithm for TSP. It provides, in polynomial time, a solution for TSP that is at most $\frac{3}{2}$ times worse than the optimal solution. This algorithm is used in the programs for two reasons: it quikly provides an upper bound, that can be useful in the real world scenario and it is also used as a "good" initial solution for the local searches in the first iteration of the multistart.
I got the code for the Christofiedes algorithm from a public repository[1], making the appropriate changes to fit it in my programs.

Simple Search The Simple Search implemented is a greedy basic local search with the following characteristics:

- The neighbour selection is done trough *first improvement* in the 2opt neighbourhood.
- The stopping criteria is one: stop when there are no better neighbours (local optimum found).

Simulated Annealing The implemented Simulated Annealing has the following characteristics:

- The next neighbour to compare is chosen at random from the 3opt neighbourhood (2opt included).
- The probability to accept a non improving neighbour is exponentially decreasing with the number of iterations and is computed as follows: $probability = e^{-((k/it\_r)*t\_desc)}$, where k is the iteration counter and it_r and t_desc are parameters for the speed of the decrease of the temperature.
- The stopping criteria are two. The first one stops the search when it reaches a maximum of consecutive non accepted neighbours, this usually means that there are not many improving neighbours and the temperature is low. The second stopping criteria is a time limit. The program doesn't check if there actually are no more better neighbour and it (rarely) may stop before reaching a local optimum. This choice was taken after some test runs and is supported by the fact that it wouldn't be computationally worth it to do this type of check on a 3opt neighbourhood at each step, where the maximum of non accepted neighbours gives a good enough stopping criteria.

Tabu Search The Tabu Search has the following characteristics:

- The neighbour selection is done with *steepest descent* (better neighbour).
- The search implements an intensification/diversification phases technique. It starts with an intensification phase with a small tabu list length and

[1] https://github.com/sth144/christofides-algorithm-cpp

using a 2opt neighbourhood until it reaches a maximum of consecutive non improving moves (compared to the best solution found so far). When this limit is reached, the diversification phase starts: the tabu length is increased and the exploration is done in the 3opt neighbourhood (2opt not included). The diversification phase ends if it finds a better solution or it reaches a maximum of diversification steps. After this phase ends, the search goes back to intensification and repeats the process.

- The two stopping criteria are: a time limit and a maximum number of overall iterations.

2.3 Results

All the tests for this second part of the exercise were done on my personal computer: Acer Swift 3. The tests have been done in two phases using two different sets of instances. The first set (learining set) was used to do parameter selection, while the second one (evaluation set) was used to do a final evaluation of the heuristic methods implemented.

Parameters selection

For the parameter selection phase, I used 9 out of the 10 instances of type "realistic graph" from part I, where the optimal value is available and can be used to compare the results of the different local searches.
The one parameter that is fixed among all of the methods is the time limit. It is set at 90 seconds, considering this time an approriate waiting time for the results in the real world scenario. In particular, I assumed that after finishing the design of a new type of board, it takes at least 90 seconds before the production of these boards will start.
After the tests over the 9 instance, the parameters values selected are as follows.

Simple Search The only parameter for the Simple Search is the time limit (90s). The program will do as many multistart iterations as it can in the time available.

Simulated Annealing The parameters for the Simulated annealing are:

- Time limit: 90s.
- Multistart: 8. The program will stop each multistart iterations after a fraction of the overall time limit in a way to give each iteration the same amount of time. The tests have shown that each iteration needs at most 10 seconds to provide a good results, so the maximum number of multistart is set to 8.
- Maximum non accepted: 50. A limit of 50 consecutive non accepted neighbours provided good results on the tests made, compared to higher or lower values.

- Iteration range: 2000 and Temperature Parameter: 10. These two parameters are used to define the temperature decrease over time (iterations).

Tabu Search The parameters for the Tabu Search are:

- Time limit: 90s.
- Multistart: 8. As for the Simulated Annealing, 8 is a number of multistart iterations that provide good resuts.
- Tabu List lenght: 7, and Tabu List Lenght for diversification: 20. High Values for the tabu lists lenghts would increase significantly the computational effort and waste precious time. For the intensification phase, 7 was shown to be a good size for the tabu list, while 20 was a good trade off between diversification and computational effort.
- Max iterations: The overall maximum number of iteration during each multistart. It is set to 3000, a large number that usually is not reached before the time runs out.
- Max number of non improving iterations: 30. Tests have shown that after 30/50 intensification steps it is unlikely to find a better solution, while it is more important to use the little time available to explore other parts of the solution space.
- Maximum diversification steps: 80. This number is good enough to do a good exploration during the diversification phase.

Results over the tested instances After parameter selection, these are the results over the instances used for testing (learning set). Table 4 compares the different results of the local searches with the optimal solution found in the part I. The solutions found that are a global optimum are marked with the symbol "*". If none of the methods reached a global optimum, the best solution is underlined.

Instance	Opt. sol.	Simple Search	Sim. Ann.	Tabu
tsp50-2	904.741	904.741*	904.741*	904.741*
tsp50-3	826.721	826.721*	826.721*	826.854
tsp50-4	860.409	860.409*	862.367	860.409*
tsp50-5	744.955	744.955*	744.955*	744.955*
tsp80-1	1679.19	<u>1679.67</u>	1690.58	<u>1679.67</u>
tsp80-2	1330.55	<u>1330.56</u>	1332.21	<u>1330.56</u>
tsp80-3	1329.72	1330.76	1330.72	1329.72*
tsp80-4	1556.56	1558.72	1570.14	<u>1557.6</u>
tsp80-5	1501.31	1501.31*	1505.05	1506.71

Table 4: Results from the learining set.

The average errors of the found solutions, when compared to the optimal solutions are:

- Simple Search: 0.027%.
- Simulated Annealing: 0.24%.
- Tabu Search: 0.05%.

All of the three methods provided very good results, with the Simple Search performing slighlty better than the other two. This result can be explained with an intuition I had during the testing phase: over all features and characteristics of the different local searches, the best one is the multistart. While the Tabu Search and the simulated Annealing use good techniques to avoid local optimum, it is very hard to explore carefully the space and find the global optimum. Apparently, the best way to find the optimal solution is doing multiple searches starting from different points. The Simple Search has no way to escape from the local optimum but it's very fast to reach it and this allows it to have more time to do more multistart iterations.

Evaluation

After the parameter selection, the final evaluation of the methods, using the selected parameters values, was made on a different and new set of instances (evaluation set). This instances are of the type "realistic graph" (from part I) and have dimensions of 100, 150 and 200 nodes, with 5 instances for each dimension for a total of 15 instances.
In this evaluation phase, the optimal solutions are not available, so the solutions found are compared with each other and, mostly, to the 1-tree lower bound. To understand how close is the lower bound to the optimal value, I computed the average error between the 1-tree lower bound and the optimal solution for each of the instances in the learning set. This test showed that, on average, the 1-tree lower bound is 19.65% lower than the optimal solution.
All of the implemented heuristic methods have some parts of randomization, so to have more valid results each instance was run 5 times. Just to show some examples, Table 5 reports the results of the best run over some selected instances (the best result among the three methods is underlined). The final evaluation, with the average errors over the 15 instances, with 5 runs for each instance, is shown in table 6.

Instance	1-tree lb	Simple Search	Sim. Ann.	Tabu
tsp100-3	1374.46	1736.06	1740.45	1734.72
tsp150-2	1872.56	2222.65	2241.44	2290.25
tsp150-4	1863.8	2176.45	2182.85	2316.74
tsp200-2	2260	2711.85	2690.75	3003.52
tsp200-3	2057.34	2448.36	2463.38	2718.8

Table 5: Some results from the evaluation tests.

Method	Average error
Simple search	19.07%
Simulated Annealing	24.4%
Tabu Search	23.3%

Table 6: Average errors from the final evalutation.

It is useful to note that, given the fact that the tests have shown an average error of the 1-tree lower bound of 19.65%, the Simple Search, with an error of 19.07%, can be (weakly) assumed to be give solutions very close to the optimal solution.

2.4 Conclusions

While all methods performed similary in the learning set, the evaluation tests gave much worst results for the Tabu search and for the Simulated Annealing with respect to the Simple Search. The main difference between the learning set and the evaluation set is the size of the instances. From this we can deduce that the parameters should be calibrated also on the size of graphs.

Additionally, these tests allows us to make some assumptions over the solution space of the problem that can explain the results obtained. We can say that the solution space has probably a low density of local optimum, otherwise the Simple Search wouldn't perform so well. In fact, in a solution space with high density of local optimums a basic local search like Simple Search may never reach a global optimum, that is not our case.

The main feature that allows the Simple Search to perform so well is the large time limit. As I stated before, the best feature seems to be the multistart but only because there is the assumption of a big time availability. In the case of little time, the Simple Search wouldn't be able to do many iterations and would just get stuck on a few local optimums, that are unlikely good. Overall, the Simple Search seems to be the method that works best in practise, but it doesn't give many certainties and may be risky to deploy it without enough tests.

Finally, for the real word case, where the large time assumption holds well, the best solution would be to use the Simple Search. Alternatively one could

prepare some good parameters values for different sizes of the problem after doing many parameter calibration phases -or develop an algorithm that does it automatically- and use the Tabu Search, which is a more stable method.